JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 19, 503-516 (2003)

Robust TCP Connections for Fault Tolerant Computing*

RICHARD EKWALL, PÉTER URBÁN AND ANDRÉ SCHIPER
École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
Distributed Systems Laboratory
CH-1015 Lausanne, Switzerland
E-mail: {nilsrichard.ekwall, peter.urban, andre.schiper}@epfl.ch

When processes on two different machines communicate, they most often do so using the TCP protocol. While TCP is appropriate for a wide range of applications, it has shortcomings in other application areas. One of these areas is fault tolerant distributed computing. For some of those applications, TCP does not address link failures adequately: TCP breaks the connection if connectivity is lost for some duration (typically minutes). This is sometimes undesirable. The paper proposes *robust* TCP connections, a solution to the problem of broken TCP connections. The paper presents a session layer protocol on top of TCP that ensures reconnection, and provides exactly-once delivery for all transmitted data. A prototype has been implemented as a Java library. The prototype has less than 10% overhead on TCP sockets with respect to the most important performance figures.

Keywords: session layer protocol, TCP, performance, fault-tolerant distributed computing, quasi-reliable channels, Java

1. INTRODUCTION

When processes on two different computers communicate, they most often do so using the TCP protocol [1]. The reasons for the popularity of TCP are threefold. Firstly, it offers a convenient interface for communication: a bi-directional byte stream. Secondly, it hides most problems of the communication channel from the programmer: message losses, duplicates and short losses of connectivity. Thirdly, it is extremely flexible and well engineered: it suits needs as different as short lived HTTP sessions, long lived file transfers, and continuous low traffic sessions like a remote login. Moreover, TCP can work on low-latency reliable local networks and on the high-latency not-so-reliable Internet with acceptable performance.

While TCP is appropriate for a wide range of applications, it has shortcomings in other application areas. One of these areas is fault-tolerant distributed computing. Many algorithms in fault-tolerant distributed computing assume so called *quasi-reliable channels*: If a process *p* sends a message *m* to process *q*, *m* will eventually be received by *q* if

Received May 15, 2002; accepted July 25, 2002.
Communicated by Biing-Feng Wang, Stephan Olariu and Gen-Huey Chen.
* Research supported by a grant from the CSEM Swiss Center for Electronics and Microtechnology, Inc., Neuchâtel and by OFES under contract number 01.0537-1 as part of the IST REMUNE project (number 65002). A preliminary version of the paper was presented at the 2002 International Conference on Parallel and Distributed Systems, Chungli, Taiwan.

neither *p* nor *q* fails [2]. An obvious way to implement quasi-reliable channels is to use a TCP connection between *p* and *q*. Unfortunately, TCP does not address link failures adequately. TCP breaks the connection if connectivity is lost for some duration, typically minutes, but the connection is broken only if TCP actually wants to send data or if keepalives are sent). This might sometimes be undesirable, and hence we need a way to recover from broken TCP connections. Potential applications that would benefit from such a feature are applications that are willing to wait for connectivity longer than the default TCP parameters allow. Examples include long-lived remote login sessions on a computer not permanently connected to the Internet (mobile devices or a PC with a modem). There are more elaborate examples from fault tolerant distributed computing (the reader not interested might skip the next paragraph). The explanation requires some additional context.

In fault-tolerant distributed computing, process failures and link failures are often abstracted using *group membership*. A group membership service offers each process a *view* of the system, the set of processes the process can currently communicate with. The view changes over time as (1) processes crash / recover, or (2) link failures occur / are repaired [3]. There are two kinds of group membership: (1) *primary partition* group membership, in which processes agree on the sequence of views, and (2) *partitionable* group membership in which multiple concurrent views can simultaneously exist. In each case, broken TCP connections can be used to trigger changes in views [3]. We shall argue here that this is not a good idea when using primary partition group membership; consequently, link failures should be transparent, and we can achieve this by robust TCP connections. Consider a replicated server with three replicas s_1, s_2, s_3. Assume a partition failure which partitions s_3 away from s_1 and s_2. If link/partition failures are transparent, nothing needs to be done when the partition failure is repaired. In contrast, if failures are not transparent, all server updates that took place during the partition failure need to be explicitly forwarded to s_3 (by s_1 or s_2). A detailed discussion of this issue can be found in [4].

Robust TCP connections present a solution to the problem of broken TCP connections. Robust TCP connections have the same interface and properties as standard TCP connections, except that these connections never break due to network problems (and thus implement the quasi-reliable channel abstraction). We define a session layer protocol on top of TCP that ensures reconnection, and provides exactly-once delivery for all transmitted data. A prototype has been implemented as a Java library (however, nothing prevents a C implementation). The prototype has less than 10% overhead on TCP sockets with respect to the most important performance characteristics: response time and throughput. Robust TCP sockets integrate seamlessly into Java. Source code integration is done by replacing occurrences of `new Socket` and `new ServerSocket` by creating instances of our replacement classes. Binary integration requires a few changes in the Java core libraries. These changes would make it possible to replace sockets with robust sockets in a Java application without re-compiling and without changing the application.

The rest of the paper is structured as follows. Section 2 discusses design issues for robust TCP connections. Section 3 presents the protocol. Section 4 discusses the implementation of robust TCP connections in Java. Performance figures are given in section 5. Related work is discussed in section 6, and section 7 concludes the paper.

2. DESIGN OF THE PROTOCOL

2.1 Requirements

In this section, we present our requirements for robust TCP connections, along with their implications for the design of the protocol.

Using robust TCP should be as transparent for the user as possible. Therefore robust TCP should have the same interface as standard TCP connections and offer the same service: a bidirectional stream of bytes. For our prototype, this implies that robust TCP should offer the Java sockets interface (integration into Java is discussed in detail in section 4.2). Our functional and non-functional requirements are as follows:

Duration of connection. In standard TCP connections, the connection is closed whenever data is sent over the connection, but no acknowledgment is received for several minutes. Robust TCP connections should only be closed if the application explicitly requests this. This means that robust TCP does not have any timeout mechanism that might lead to breaking the connection. Specifically, robust TCP connections must survive link failures and network partitions.

Flow/congestion control. Robust TCP should have the same flow control / congestion control mechanisms and behavior as TCP. It should use buffers of limited size.

Performance. Robust TCP should incur an overhead of less than 10% on normal operation with respect to all relevant performance figures: response time and throughput. The overhead on the time to open/close connections is less important, as robust TCP connections are long-lived. Also, the overhead on the network should be a small fraction of the overall traffic.

Ease of deployment. The implementation should be lightweight and deployment should be easy. Robust TCP will require extensions at both endpoints of the connection, as losses of connectivity affect both sides; modifying just one endpoint is not sufficient. However, we do not want to rely on daemons supporting our protocol. Moreover, we want a user space implementation, with no need to modify the kernel or to have administrator privileges.

The ease of deployment requirement implies that we can modify neither TCP nor any of the lower layers, nor can we configure the parameters of these layers.

Modifications to the TCP kernel code would be very small and would essentially mean changing a timeout value from some number of minutes to $timeout = \infty$. This would guarantee that TCP never closes connections due to a timeout.

However, modifying TCP has several disadvantages. First of all, it would require modifications to kernel code. This would, of course, reduce the portability of the code since many different TCP stack implementations exist and would all need to be modified. For non-open-source platforms, the integration of robust TCP would be complicated, if possible at all.

Secondly, the user of the protocol would have to re-compile the kernel in order to be able to use robust TCP. The average user is not necessarily comfortable with this and

most users will not blindly trust new kernel code (an unintentional or malicious misbehavior can never be excluded).

For these reasons, we decided to implement a protocol on top of TCP, in the session layer of the OSI reference model (Fig. 1). The purpose of the protocol is to deal with broken TCP connections. Most performance requirements are easily fulfilled if the overhead of the session protocol is always just a small fraction of the traffic generated by TCP. We discuss performance issues in more detail in section 5.

In addition to the standard TCP interface, the application might want to be informed about the state of session connections (robust TCP connections do not need to be the same black box to the application as standard TCP connections). We plan to extend the interface to provide the following information: the number of bytes sent but not acknowledged, the time elapsed since the last send operation whose data was not acknowledged, and the duration for which a receive (or send) operation has been blocked.

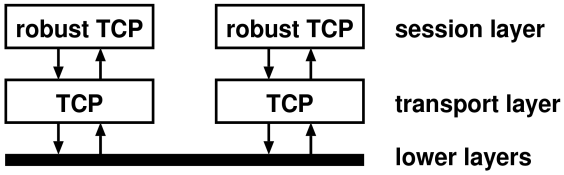

Fig. 1. The robust TCP protocol in the OSI reference model.

2.2 Issues at the Session Layer

TCP will close a connection if two hosts cannot contact each other for several minutes and data is being exchanged. When this happens, the session protocol (1) must reconnect the two parties, (2) must be able to uniquely identify a connection, and (3) must ensure that all data sent is received exactly once.

The issues related to reconnection are the following. First of all, the client (the party that did an active open) must initiate the reconnection to the server (the party that did a passive open), and not vice-versa. The reason is simply that only the server has a static address to connect to (furthermore, server to client connections are problematic if the communication parties are separated by firewalls). This implies that the server cannot close the socket on which it listens for TCP connections when it is no longer willing to accept new connections. This socket needs to remain open as long as there are active session layer connections. The second issue is that the reconnection attempt might fail. In this case, the client should repeatedly try reconnecting.

A session layer connection is potentially associated with multiple transport layer connections (Fig. 2). This means that we need to identify the session layer connection upon reopening a TCP connection. A session layer connection is uniquely identified by the combination of (1) the IP address and the port number of the TCP socket on the server side, along with (2) a unique connection identifier (CID) generated by the robust TCP server (session layer) upon the first connection attempt. This also allows us to distinguish a reconnection attempt from the first connection attempt of a new session layer connection.

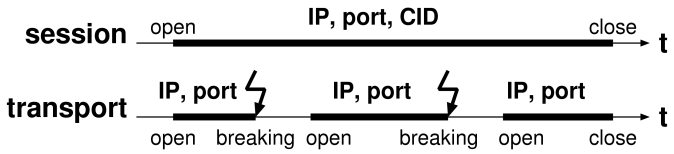

Fig. 2. Lifetime of session and transport layer connections.

When a TCP connection is broken, we do not know how much data has been successfully transmitted (we cannot access the information in TCP acknowledgments). Therefore all transmitted data must be buffered and retransmitted upon reconnection if necessary. As the protocol should only use a buffer of limited size, it has to exchange *control messages* to acknowledge received data. Upon receipt of such a message, a part of the buffer can be discarded. We must make sure that the acknowledgments constitute only a small portion of the overall traffic generated by the connection. Also, flow control issues arise if the buffer fills up.

2.3 The Problem of Control Messages

The control messages can be passed between the client and the server in two ways. Either the messages are passed in-band, multiplexed with application data, or out-of-band, on a different channel. We have chosen an out-of-band solution, primarily because the in-band solution poses severe performance problems, and secondarily because it is more complex. To understand why, let us first explain how an out-of-band solution could be implemented. The idea is simple. The session layer *send* operation buffers outgoing data, and the session layer *receive* operation sends acknowledgments. Also, a lightweight flow control mechanism is needed to block the *send* operation as long as the outgoing data buffer is full.

Let us contrast this implementation with the in-band solution. The problems are as follows:

- Multiplexing and demultiplexing two streams may be costly by itself, especially if data is transmitted in small chunks. This is the easiest problem; a solution similar to Nagle's algorithm [5] could offer acceptable performance.
- The data stream and the stream of control messages are independent. Even if no data is sent to one of the communication parties, that party may still receive control messages. For this reason, each party has to constantly read the TCP stream to check for control messages. This requires a dedicated *control* thread to read the socket. So, when data arrives, it has to pass through the control thread before reaching the application thread that reads the socket. This leads to context switching and one extra copy of the data to an intermediate buffer, and yields poor performance.
- The solution requires a rather complex flow control mechanism. If the control thread receives a lot of data and the application is not ready to receive data, the intermediate buffer fills up. The control thread must continue reading in order not to miss control messages. This implies that the protocol has to discard any further data and has to ask the other side for retransmission. In contrast, an out-of-band solution needs only retransmission of data when the TCP connection breaks.

The question remains as to how to pass out-of-band control messages. The two choices are (1) UDP datagrams and (2) a separate TCP connection. We chose the UDP solution for reasons of performance and resource utilization. Indeed, the TCP solution needs twice as many TCP connections and TCP ports on each side (2 per session layer connection). In contrast, a server can share the UDP control port among all the connections it manages. Also, the TCP solution exchanges more IP packets during the whole lifecycle of the connection (open, data transfer and close).

The UDP solution might seem more complex at first: (1) We need to identify the connection in each control message, and (2) we need to ensure reliable delivery and FIFO order of control messages (we can afford to lose some acknowledgments, though). However, the TCP solution would result in an equally complex implementation, as it would have to ensure reliable delivery and FIFO order as well (in case the TCP connection for control messages breaks).

Finally, note that the TCP solution should be preferred if the connection passes through a firewall, as firewalls are usually configured to reject UDP packets. However, this was not a problem for us, and if the need arises, the protocol can be easily modified to use TCP.

3. THE SESSION LAYER PROTOCOL

A robust TCP session has three phases: (1) connection establishment (opening), (2) data exchange, and (3) connection termination (closing). Whenever TCP errors occur, the protocol enters the reconnection phase. We now describe each of these phases in detail, and then discuss how TCP errors are handled.

3.1 Opening a Connection and Reconnection

The client starts by establishing a TCP connection with the server, sends (over the TCP connection) the *new connection* control message together with the number of the UDP port used for exchanging control messages (Fig. 3), and then waits for a unique connection identifier (CID) from the server. The server assigns CIDs in the order k, $k + 1$, $k + 2$, etc. k is chosen randomly when the server starts up, in order to avoid that clients of a server that used to listen on the same port (and that quit or crashed) confuse the server (One faces a similar issue when choosing initial TCP sequence numbers.)

The CID is always chosen to be different from 0, in order to give a special meaning to 0. It is used to distinguish a new session from an existing session that re-opens its TCP connection. If the server accepts the connection, a new unique session identifier is sent to the client, together with the server UDP control port. The details of the protocol are shown in Fig. 3.

New robust TCP sessions can be refused. This happens when the server has closed the session layer socket, or when it temporarily runs out of resources (too many open connections). In that case, the server sends a *refusal* (0) control message (instead of a non-null connection identifier) and a reason code. The client will try to reconnect if the reason code indicates a temporary reason for refusal.

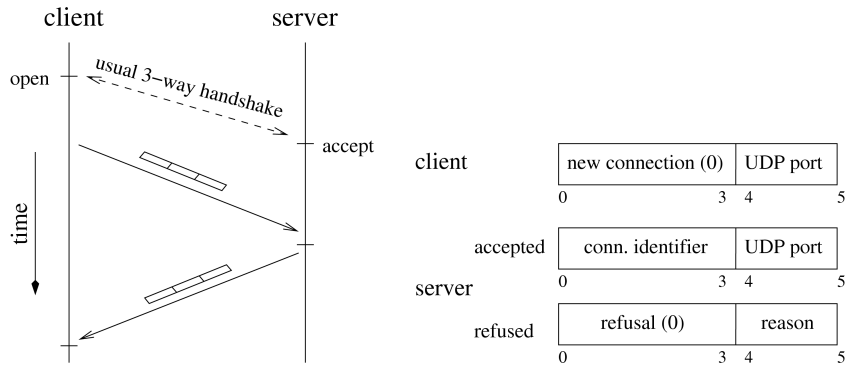

Fig. 3. Opening phase.

If the TCP connection breaks during data exchange, re-establishing the TCP connection uses a similar protocol, shown in Fig. 4. The client opens a new TCP connection to the server, then sends the session layer connection identifier and the number of bytes received in this session (we use a 32 bit counter that wraps around to 0 when it reaches 2^{32}). The server answers by re-sending the connection identifier to confirm that the re-connection has been accepted and the number of bytes received. As soon as any of the parties knows how many bytes have been received by the other party, it starts retransmitting data that was lost. Finally, the session is ready again for exchange of data. To the user, the session appears to be open during the whole reconnection process, i.e., send and receive calls return normally.

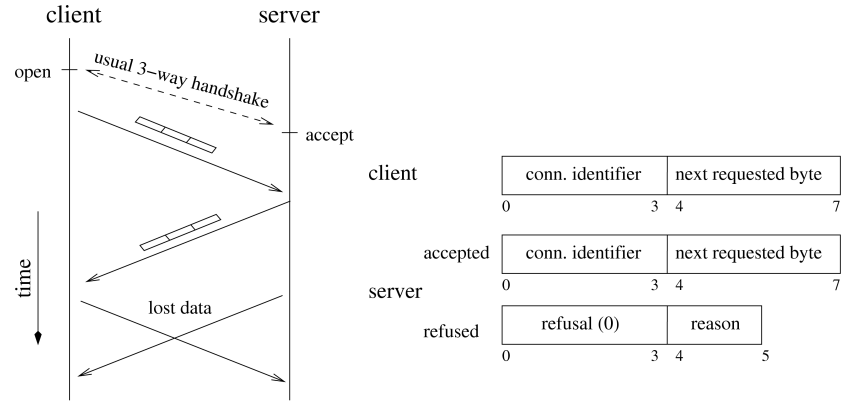

Fig. 4. Reconnection phase.

3.2 Data Exchange

Anytime the user writes some data on the robust TCP connection, the data is stored in a buffer and then sent over the TCP connection. The protocol also maintains a counter for the total number of bytes sent over the robust TCP connection. Whenever an acknowledgment is received, acknowledged data is removed from the buffer. The ac-

knowledgment, similar to TCP's acknowledgments, points to the next byte that the reader process is expecting to receive. The exact format of acknowledgments and other control messages is discussed later.

The buffer for outgoing data is of limited size, hence some flow control is needed. When the buffer fills up, the protocol blocks send operations. Also, it sends a control message that forces the other party to acknowledge data. We set the default parameters such that send operations hardly ever block; acks are sent after every 8 kbyte received and the outgoing buffer is of size 16 kbyte (these parameters can be configured by the user). Also note that the default setting yields few acknowledgment packets compared to the number of data packets.

3.3 Closing the Connection

We only discuss how the session is closed; handling of half-close is analogous. Closing the session is done by closing the TCP connection; control is returned immediately to the application. However, the two sides have to notify each other that the connection is closed in order to free up resources associated to the connection. This notification is done asynchronously using control messages. So the UDP control ports are active until these notifications have been exchanged.

If the server closes the session and the client still wants to use it, TCP generates an error. It is possible that the server could not notify the client about the closing of the session at this point. In this case, the client will try to reconnect, and the server implicitly notifies the client about the closing of the session by simply rejecting the reconnection attempt.

3.4 Handling TCP Errors

Whenever a TCP socket operation returns an error, the protocol first tries to gracefully close the TCP socket. Then, if the error occurred during the opening phase, the opening phase is restarted. Otherwise, the protocol enters the reconnection phase – or re-enters the reconnection phase if the error occurred in the reconnection phase.

The protocol allows increasing the delay between two consecutive reconnection attempts using an exponential backoff strategy. Furthermore, some TCP errors indicate the failure of the other party rather than the loss of connectivity [6]. We could use this information to avoid unnecessary reconnection attempts.

3.5 UDP Control Messages

The structure of the UDP control messages is presented in Fig. 5. The message starts with eight header bytes. The first four bytes of this header store the identifier of the connection. The next byte defines the type of the message: ACK to acknowledge data, REQ_ACK to force the other side to send an acknowledgment, and CLOSE, HALF_CLOSE and RESET to manage closing connections. Since UDP does not guarantee delivery, control messages that need to be acknowledged (such as a CLOSE message initiating a three-way handshake) are flagged using the flag byte. The header ends with a two bytes identifier for the control message identifier. These two bytes have a meaning only if the control message needs to be acknowledged, as specified by the flag byte.

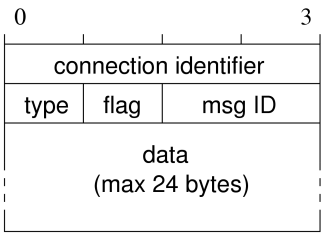

Fig. 5. Structure of control messages.

An acknowledgment control message (type CONF) carries this identifier to indicate that the message has been received. Finally, the message ends with a data section. For acknowledging data (ACK and REQ_ACK) four bytes represent the number of bytes received. The other control messages have no data section.

Note that we could have compressed the data in the control messages. However, this is not worthwhile as the overhead is negligible compared to the overhead of UDP, IP and the lower layers. In particular, control messages easily fit into the smallest Ethernet frame.

4. JAVA IMPLEMENTATION

4.1 Classes

The implementation of robust TCP provides the same interface as the sockets in java.net. For a smooth integration into existing programs, the classes implementing robust TCP sockets, i.e., RSocket (client side) and RServerSocket (server side), extend the classes Socket and ServerSocket of the standard Java TCP interface. The slight differences between the client and server side of a connection (in the reconnection procedure) are handled by a class that extends RSocket and implements the server side specificities (e.g., the reconnection procedure and the notifications that complete a close). This class is package-private and is instantiated only by the RServerSocket whenever a new connection is created.

Each of endpoints also needs a dedicated thread that constantly reads control messages from the UDP socket. All connections in the same JVM (both client and server side) share this UDP socket. This means that a single control thread is used to read the control messages and dispatch them to the right connection. Using few threads is essential for achieving good performance on the server side.

4.2 Integration Into Java

Since the robust TCP sockets extend the Java sockets, the user can simply replace any call to the constructor of Socket or ServerSocket by a similar call to RSocket and RServerSocket. This is a rather easy way to integrate robust TCP connections into new applications or code available in source code form.

Let us now discuss how to integrate robust TCP connections into existing applications without changing the source code. Java provides a way to use modified sockets instead of the standard ones without modifying or recompiling the application. The user of the Java libraries can call the method ServerSocket.setSocketFactory (for the server side) and Socket.setSocketImplFactory (for the client side) with as parameter an object that will serve as a factory for socket implementations. Socket implementations extend the SocketImpl class. Similarly C sockets, this class provides a client interface to connect to a remote host (bind and connect), and a server interface to accept connections (bind), listen and accept).

Unfortunately, Java socket factories are not flexible enough to allow the integration of robust TCP sockets (that is, without modifying the Java core libraries). The methods setSocketFactory and setSocketImplFactory can only be called once in an application and no plain Java sockets can be created after the call. This is a problem for us, as we access TCP by plain Java sockets. However, several requests are present in the Java bug tracking database [7] that aim to make the socket factory features more flexible. Once an improved socket factory framework is released by Sun, we will be able to achieve fully transparent integration of robust TCP sockets into existing code. The integration wll not require any modifications to the Java core libraries (java.net).

5. PERFORMANCE

The benchmarks used to measure the performance of the robust TCP sockets are taken from IBM's SockPerf socket micro-benchmark suite, version 1.2 [8]. These experiments do not benchmark all aspects of communication with sockets. Nevertheless, they should give an indication of the overhead of the robust TCP sockets with respect to the plain TCP sockets. The benchmarks are as follows:

TCP_RR: A message (request) is sent using TCP to another machine which echoes it back (response). The TCP connection is set up in advance. The results are reported as a throughput rate of transactions per second, which is the inverse of the request / response round-trip time. The benchmark is repeated several times with different message lengths. The default length in SockPerf is one byte.

TCP_STREAM: A continuous stream of messages is sent to another machine, which continuously receives them. The results are bulk throughputs in kilobytes per second. The benchmark is run with several different message lengths. The default message length in SockPerf is 8 kbytes.

TCP_CRR: First, a connection is established between the two machines (connect). Then, a message (request, by default 64 bytes) is sent using TCP, and is replied to (by default 8 kbytes). This reflects the message size of a typical HTTP query. The costs included in the benchmarks are those of the connection establishment, the data exchange and the closing of the connection.

The benchmarks were run with two PCs running Red Hat Linux 7.2 (kernel 2.4.9). The PCs have Pentium III 766 MHz processors and 128 MB of RAM, and are interconnected by a 100 Base-TX Ethernet. The Java Virtual Machine was Sun's JDK 1.4.0.

The results, as well as the relative performance of the robust TCP sockets versus Java sockets, are summarized in Table 1 and Fig. 6. They show that the overhead of the robust TCP sockets over Java sockets is low (5.7% and 1%) for the TCP_RR and TCP_Stream tests, except for small message lengths in the TCP_Stream test (we are working on optimizing this case). The overhead for these tests is probably due to (1) the one extra copy of transmitted data into the retransmission buffer at the session layer, and (2) the control message processing.

Table 1. Java vs. robust TCP in the three benchmarks.

Benchmark	Robust TCP	Java TCP	Overhead
TCP_RR	8572 tr./s	9061 tr./s	5.7%
TCP_Stream	10785 kB/s	10889 kB/s	0.95%
TCP_CRR	3.34 ms	1.30 ms	157%

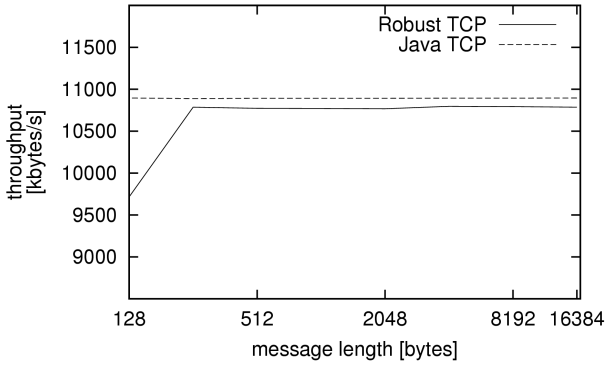

Fig. 6. Results of the TCP Stream benchmark.

The TCP_CRR test shows a bigger overhead. The overhead is due to the message exchange upon opening the connection (see Fig. 3). However, this benchmark measures the performance of short-lived TCP connections, whereas robust TCP connections only make sense for long-lived connections – short-lived connections are not likely to break. For this reason, we did not put any effort into optimizing for the TCP_CRR benchmark. A possible optimization is to wait for the first data packet such that the session layer messages can be piggybacked.

6. RELATED WORK

We start with papers about fault-tolerant TCP connections. Zhang and Dao describe *persistent connections* [9], which can recover from broken transport layer connections, just like our robust connections. As ours, their prototype is also implemented in a library

on top of sockets. However, Zhang and Dao have a more ambitious goal and do not meet our requirements. Zhang and Dao provide connections where the transport layer endpoints might change their location and/or identity. For example, one endpoint might be a mobile device that migrates, or a process that crashes and then recovers. As the goal is more ambitious, the solution is more complex. It involves an addressing scheme distinct from TCP addressing and a name service used to store information about endpoints. On the other hand, data loss is possible if a connection breaks in an unanticipated manner, while our protocol avoids this. The authors did not avoid data loss because they focused on connections that break due to process crashes, rather than network problems. In such a setting, a session layer mechanism is not enough to provide exactly-once delivery; some help is needed from the application.

The FT-TCP protocol [10], and STCP [11] to some extent, also aim to make TCP connections fault-tolerant to the crash of one endpoint (while our protocol makes the connection fault-tolerant to link failures). After the crash, either another node has to take over the connection, or the failed node has to recover. Even though the problem is different from ours, the solutions involve a lot of common tasks: buffering data and synchronizing the new node to the state of the stream with the help of the buffered data. A difference is that FT-TCP and STCP require changes in the kernel, as they augment or modify the transport layer. An interesting point in FT-TCP is that the other (non-fault tolerant) endpoint of the connection runs TCP without any changes; we cannot provide this property, though, as a broken connection affects both endpoints.

The protocols in [12, 13] adapt TCP to wireless environments. Connectivity can be lost in such environments for a long time. The solutions usually passivate the TCP connection when connectivity is lost to avoid the condition that TCP reacts to this condition by reducing the size of its congestion window or by breaking the connection. These protocols necessitate changes in the kernel.

Finally, let us mention that the session layer in the ISO/OSI reference model [14] offers some functionality to re-establish broken transport layer connections. The communicating parties can put *synchronization points* into the session layer stream, and it is possible to recover the state of the stream at these synchronization points later. It is the application's responsibility to set synchronization points and to buffer data that might need to be retransmitted. Our solution accomplishes exactly these tasks, making synchronization and buffering transparent to the application.

7. DISCUSSION

We presented robust TCP connections for fault tolerant distributed computing. Robust TCP connections, unlike TCP connections, address link and partition failures in a manner adequate for a range of applications. Robust TCP connections never break if connectivity is lost.

We implemented robust TCP connections as a session layer protocol on top of TCP that ensures reconnection, and provides exactly-once delivery for all transmitted data. Our Java prototype has less than 10% overhead on TCP sockets with respect to the most important performance features. It can be easily integrated into existing applications.

As future work, we plan to extend the standard TCP interface in order to provide information about the state of session connections to the application. Useful information

includes the number of bytes sent but not acknowledged, the time elapsed since the last send operation whose data was not acknowledged, and the duration for which a receive (or send) operation has been blocked. Yet another idea is to add an operation that allows the application to passivate the connection when there is no need to send data over a long period. A passivated connection uses fewer resources; in particular, the associated TCP connection would be closed.

REFERENCES

1. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, 1994.
2. A. Basu, B. Charron-Bost, and S. Toueg, "Simulating reliable links with unreliable links in the presence of process crashes," *International Workshop on Distributed Algorithms* (*WDAG '96*), LNCS 1151, 1996, pp. 105-122.
3. B. Charron-Bost, X. Défago, and A. Schiper, "Broadcasting messages in fault-tolerant distributed systems: the benefit of handling input-triggered and output-triggered suspicions differently," Technical Report IC/2002/020, EPFL, 2002.
4. G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *Computing Surveys*, Vol. 4, 2001, pp. 1-43.
5. J. Nagle, "RFC 896: Congestion control in IP/TCP internetworks," 1984, Internet Request for Comments.
6. N. Neves and W. Fuchs, "Fault detection using hints from the socket layer," in *Proceedings of 16th Symposium on Reliable Distributed Systems* (*SRDS '97*), 1997, pp. 64-71.
7. Shortcomings of SocketImplFactory, Bug report on Sun's Java Developer Connection site, 1999, http://developer.java.sun.com/developer/bugParade/bugs/4245730.html.
8. IBM Corporation, SockPerf: A Peer-to-Peer Socket Benchmark Used for Comparing and Measuring Java Socket Performance, 2000, http://www.alphaWorks.ibm.com/aw.nsf/techmain/sockperf.
9. Y. Zhang and S. Dao, "A persistent connection model for mobile and distributed systems," *4th International Conference on Computer Communications and Networks*, 1995, pp. 300-305.
10. L. Alvisi, T. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, "Wrapping server-side TCP to mask connection failures," in *Proceedings of 20th Annual Joint Conference of the IEEE, Computer and Communications Societies* (*Infocom*), 2001, pp. 329-337.
11. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson, "RFC 2960: Stream control transmission protocol," 2000, Internet Request for Comments.
12. K. Brown and S. Singh, "M-TCP: TCP for mobile cellular networks," *ACM Computer Communication Review*, Vol. 27, 1997, pp. 19-43.
13. K. Ratnam and I. Matta, "WTCP: An efficient transmission control protocol for networks with wireless links," in *Proceedings of 3rd IEEE Symposium on Computers and Communications* (*ISCC '98*), 1998, pp. 74-78.
14. ISO, Information Technology-Open Systems Interconnection-Connection-Oriented Session Protocol: Protocol Specification, ISO/IEC 8327-1, International Organization for Standards, 1996.

Richard Ekwall is a research and teaching assistant at the Distributed Systems Laboratory at the Swiss Federal Institute of Technology in Lausanne (EPFL). He received his M.S. degree in Computer Science from the EPFL in 2002 and spent the 1999-2000 academic year at the Carnegie Mellon University, Pittsburgh (PA). He currently takes part in the IST-REMUNE project. His research interests include distributed systems, fault tolerance and protocol specification and development using SDL.

Péter Urbán is a research and teaching assistant at the Distributed Systems Laboratory at the Swiss Federal Institute of Technology in Lausanne (EPFL). He received his M.S. degree in Computer Science from the Budapest University of Technology and Economics. From 1997 to 1998, he worked at the CERN European Laboratory for Particle Physics in Geneva, Switzerland. His research interests include distributed systems, middleware, fault tolerance and performance evaluation.

André Schiper has been professor of Computer Science at the EPFL (Federal Institute of Technology in Lausanne) since 1985, leading the Distributed Systems Laboratory. During the academic year 1992-93 he was on sabbatical leave at the University of Cornell, Ithaca (NY). His research interests are in the areas of fault-tolerant distributed systems, middleware, group communication, and recently mobile ad-hoc networks (Swiss MICS project). He took part in the following European projects: ESPRIT BROADCAST (1992-1995, 1996-1999), ESPRIT R&D Open-DREAMS I and II (1996, 1997-1999), IST REMUNE (2001-2003). He is member of the IST Network of Excellence in Distributed and Dependable Computing Systems (CABERNET). From 2000 to 2002, he was the chair of the steering committee of the International Symposium on Distributed Computing (DISC).